Synchronization for Multiprocessor Architectures

Most slides adopted from David Patterson

David Andrews
Computer Engineering Group
University of Paderborn

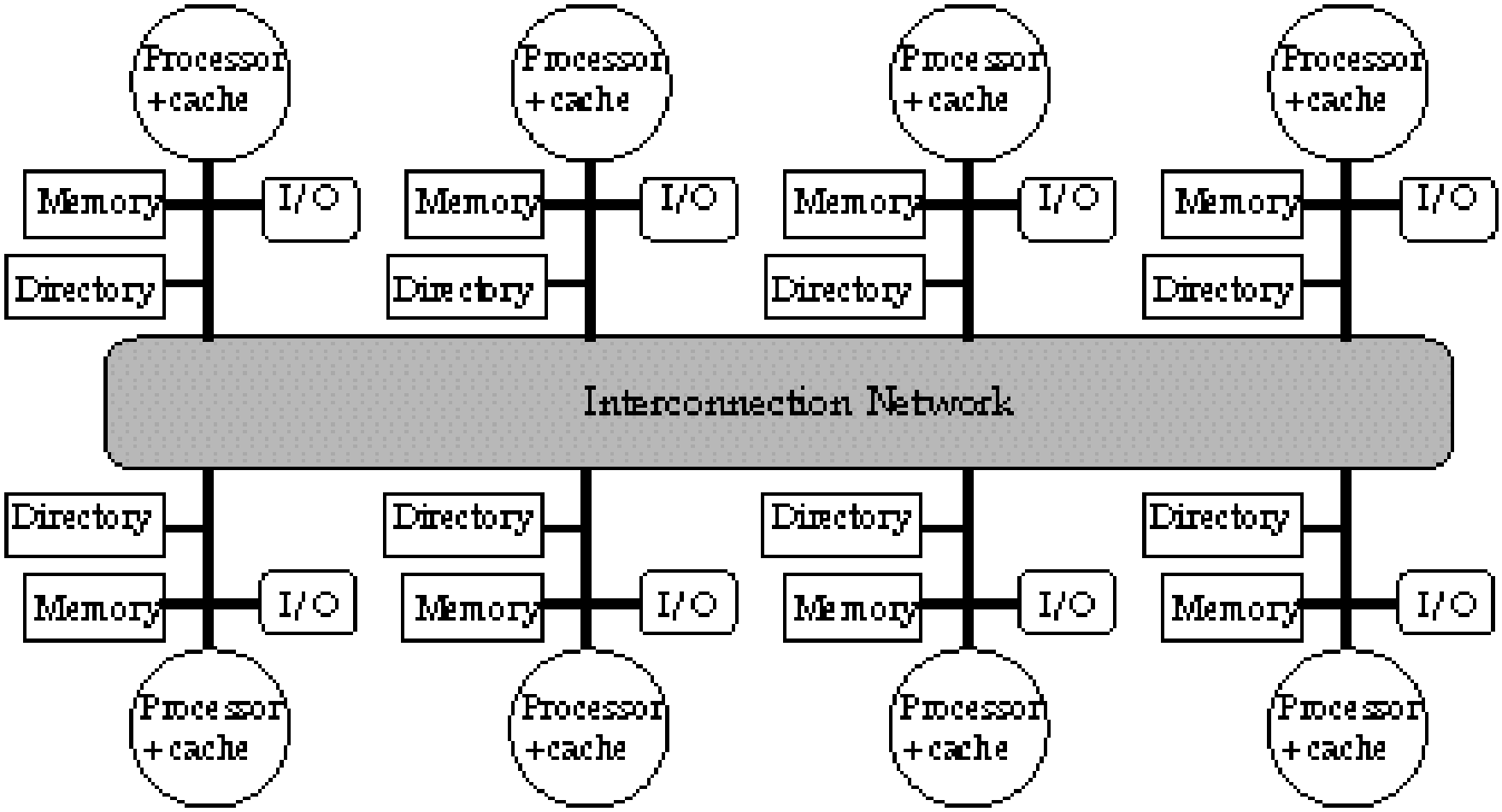
dandrews@ittc.ku.edu

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

SMP—Shared Memory Organization

- Caches serve to:
  - Increase bandwidth versus bus/memory
  - Reduce latency of access
  - Valuable for both private data and shared data
- I/O & Memory Global Access

Distributed Directory MPs



Synchronization Basics

- Build With User-Level Software Routines
  - Policy Given by API
  - Mechanism within library routine
- Two Flavors, Shared Memory and Message Passing

  - Shared Memory Synchronization Through Semaphores
    - Mutex := Simple Binary Semaphore {0,1}
    - pthread_mutex_lock(mutex)
      - Will block calling thread if M[mutex] =0
      - Will allow thread to continue if M[mutex] = 1;
        - » Will also set M[mutex] = 0; (show locked)

  - Message Passing
    - int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

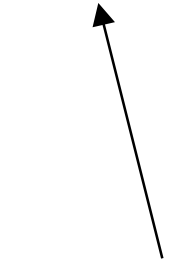
Programmers Perspective

- Mutex is "mutual exclusion".
  - Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- A mutex variable acts like a "lock" protecting access to a shared data resource.
  - Only one thread can lock (or own) a mutex variable at any given time. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.
- Mutexes can be used to prevent "race" conditions.
  - An example of a race condition involving a bank transaction is shown below:

Thread 1	Thread 2	Balance
Read balance: $1000		$1,000
	Read Balance: $1,000	$1,000
		$1,000
Deposit $200		$1,000
Update Balance $1000 + $200 = $1,200		$1,200
	Update Balance $1000 + $200 = $1,200	$1,200

Using the mutex

```
Thread1(        ){                          Thread2(    ){

Pthread_mutex_lock(&gate)                   Pthread_mutex_lock(&gate)

Read Balance                                Read Balance
Balance += deposit                          Balance += deposit

Pthread_mutex_unlock(&gate)                 Pthread_mutex_unlock(&gate)
```

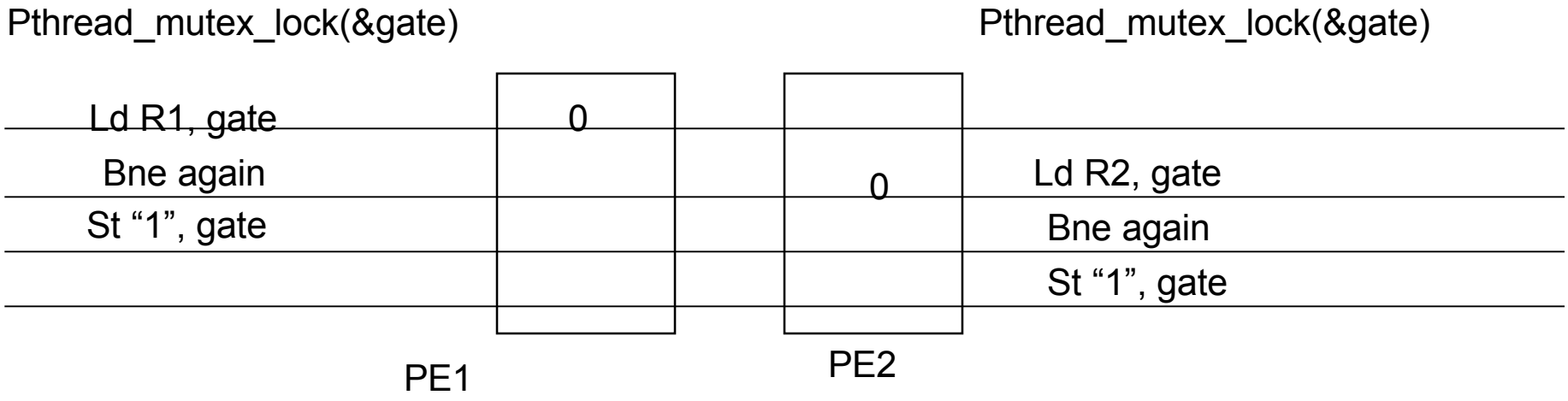


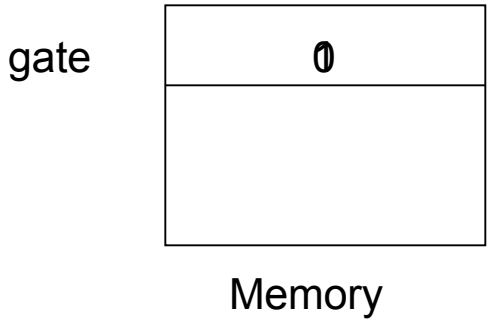
Critical Region

Synchronization

- Issues for Synchronization:
  - Uninterruptible instruction to fetch and update memory (atomic operation);
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Problem for Multiprocessors





Both Threads Enter The Critical Region !

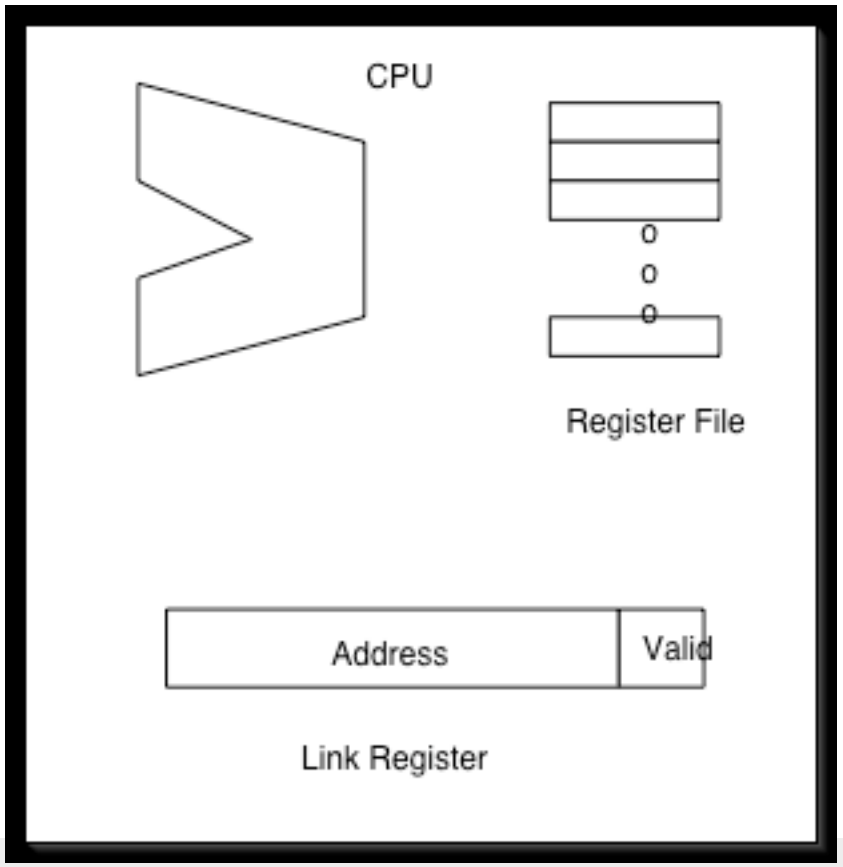
Uninterruptible Instruction to Fetch and Update Memory

- Atomic exchange: interchange a value in a register for a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    0 if you succeeded in setting the lock (you were first)
    1 if other processor had already claimed access
  - Key is that exchange operation is indivisible
- Original Atomic Instructions:
  - Test-and-set: tests a value and sets it if the value passes the test
  - Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - Guaranteed Atomicity by locking bus
    - Very poor performanceon a shared bus system

Uninterruptible Instruction to Fetch and Update Memory

- Modern Version
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceeding load) and 0 otherwise

```
LL(Rx,Ry) {
    Rx <- Mem[Ry]  sem value from memory
    link_reg <- Ry    put address into link reg
    Valid <- 1        set valid bit
}

SC(Rx,Ry) {
    if(Ry == link_reg && valid == 1)
        Mem[Ry] <- Rx
        Rx <- 1
    else
        Rx <- 0
}
```



Examples

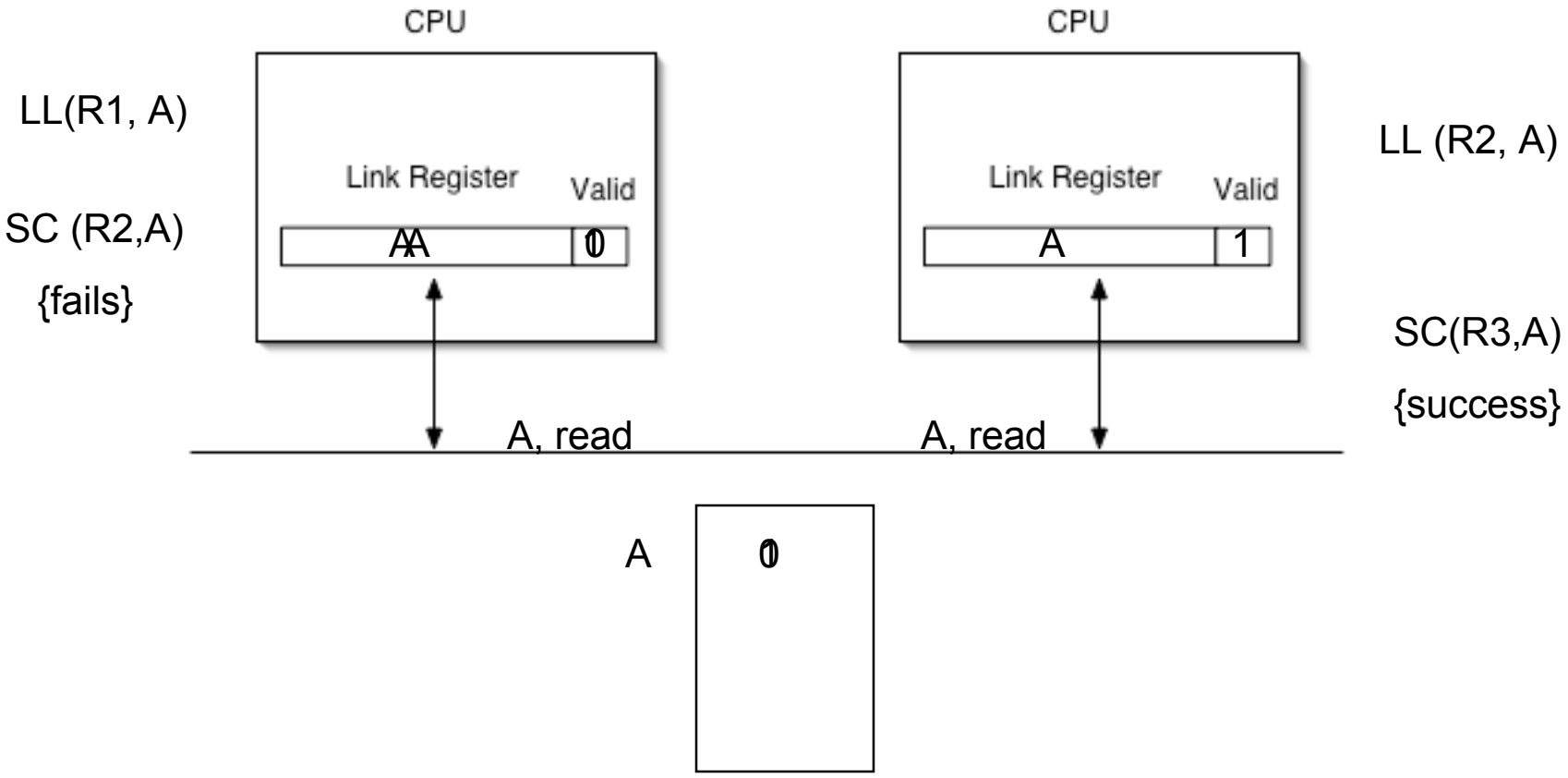
- Example doing atomic swap with LL & SC:

```
try:    mov     R3,R4           ; mov exchange value
        ll      R2,0(R1)  ; load linked
        sc      R3,0(R1)  ; store conditional
        beqz    R3,try          ; branch store fails (R3 = 0)
        mov     R4,R2           ; put load value in R4
```

- Example doing fetch & increment with LL & SC:

```
try:    ll      R2,0(R1)  ; load linked
        addi    R2,R2,#1        ; increment (OK if reg–reg)
        sc      R2,0(R1)  ; store conditional
        beqz    R2,try    ; branch store fails (R2 = 0)
```

Architecture Support



Architecture Support

- Link Register Doesn't Typically Sit on Bus
  - That's What the Cache is For !
  - Ties into Snoopy Cache Lines to Monitor Address
- Snoopy Cache Also Eliminates Bus Saturation
  - For Spin Locks, We Just Keep Trying…..
    - First Load Linked Actually a Read Miss
      - Address Stored in Both Cache and Link Register
        - » Address marked as shared in cache
      - SC is Write
        - » If some one else read SC doesn't actually happen (valid == 0)
        - » Subsequent LL reloads are read from cache
        - » If no-one else attempted to read then SC goes to Cache
        - » Now Marked as Exclusive
        - » Invalidates everyone else's cache copy
        - » Subsequent reloads using LL cause new value to be written back and cache will be updated with new value

Interesting Performance Issues

- Suppose 5 “threads” waiting for value to change (release)
  - Lock gets set back to “0” (in cache) What happens ?
    - Exclusive in owner, invalidates in the 5 waiting caches
    - First requestor causes write back and update
      - Second requestor can cause invalidate of first requestors link_reg
      - Third requestor can cause invalidate of second req’s link_reg
      - Fourth requestor can cause invalidate of third req’s link_reg
      - Fifth requestor can cause invalidate of fourth req’s link_reg
      - Hmmm……hardly seems fair to be quick….
    - Also, can cause poor performance through starvation
  - For these reasons, blocking semaphores are used
    - Can control “release” order
    - Also eliminates massive bus activity when semaphore is released

Image Processing Example

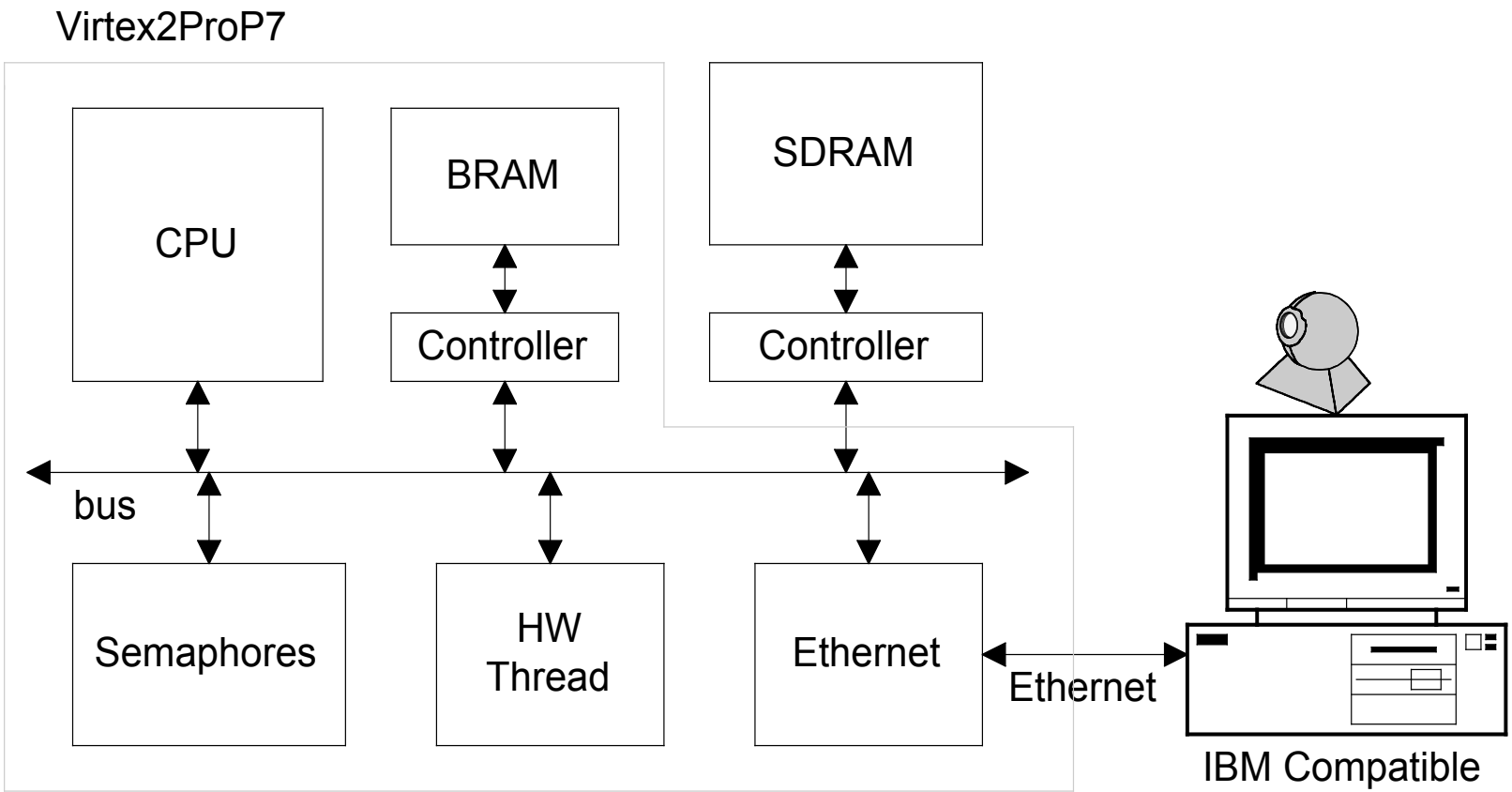
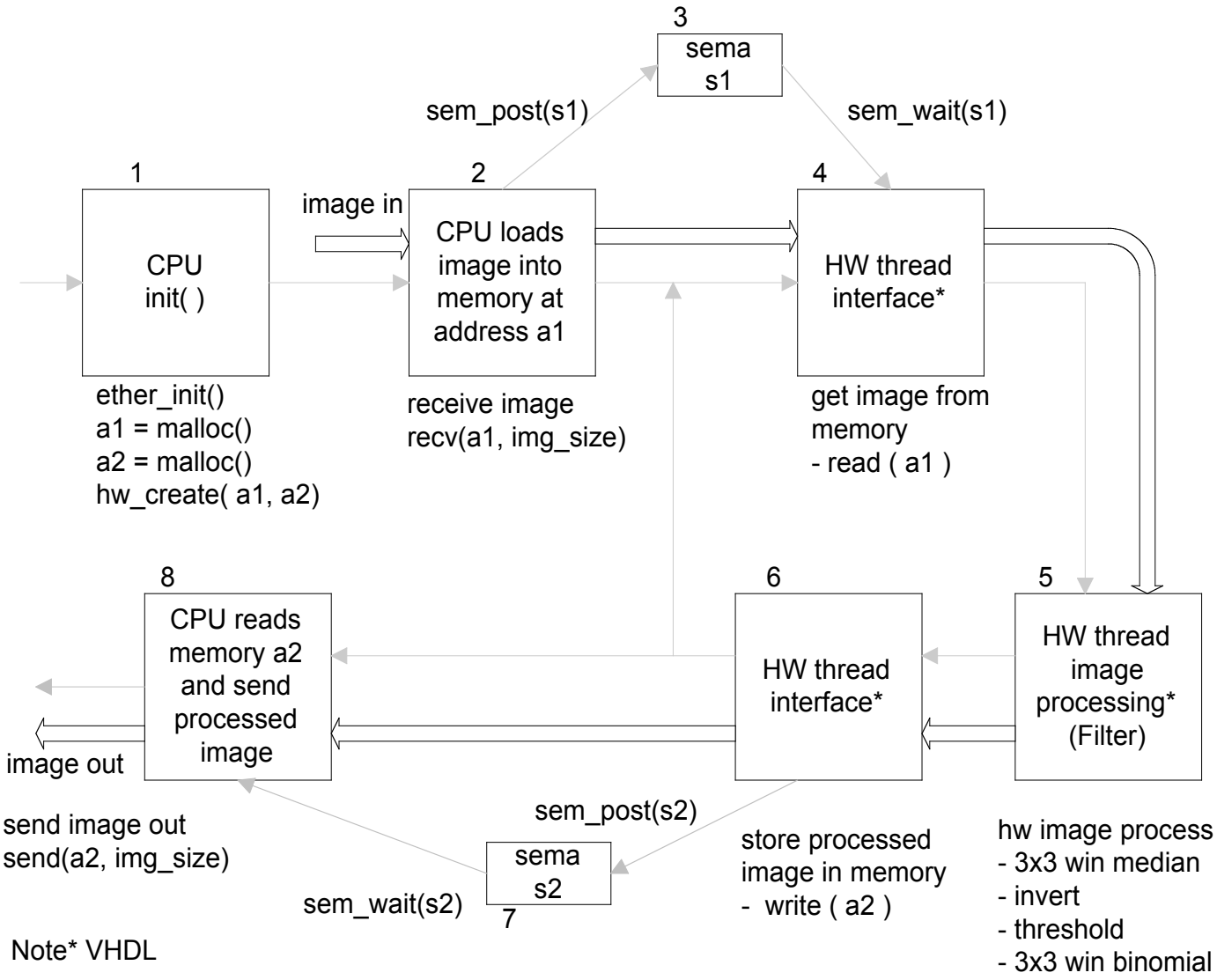


Image processing flow

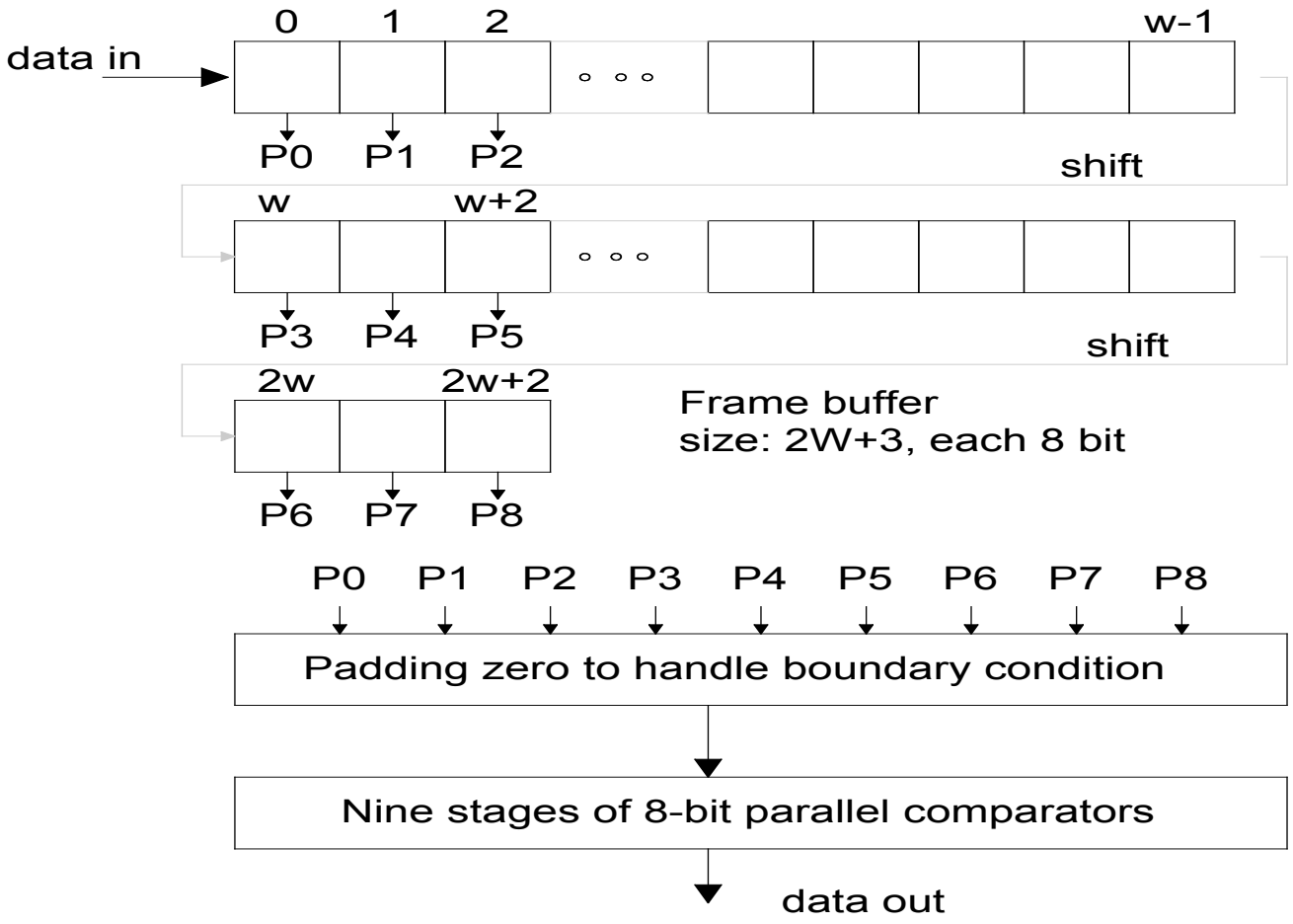


Thread Structure

```
main( ){
//Initialize Ethernet link
ether_init( );

//Raw image data pointers
        address1 = malloc(image_size)

// Processed image data pointer
        address2 = malloc(image size)

        img->in   = address1;
        img->out = address2;

//Hardware thread create API
        hw_thread_create(address1, address2, algorithm )

while (1) {
 // Get image from ethernet
    receive(img->in, img_size)

// Let hw thread know image data is available
        sem_post( S1 );

//Wait for hw thread finish processing
        sem_wait( S2 );

// Send processed image
        send(destination, img->out, img_size);    } }
```

```
                If command == run

SW:     sem_wait(S1)

        //get image from main memory
RD:     read data (address1)

        // wait for transformed image
        process wait

         // store mage to main memory
        write data(address2)

        if count != image_size
                RD:
        else
                SP:

SP:     sem_post (S2)

        branch SW
```

Hardware Median Filter



HW vs.SW Image Processing

- Image frame size 240 x 320 x 8 bits
- FPGA & CPU clocked at 100 MHz
- FPGA can process 100 frames/sec, speed-up about 40x
- Performance advantages of custom hardware

Image Algorithms	HW Image Processing	SW Image Processing Cache OFF	SW Image Processing Cache ON
Threashold	9.05 ms	140.7 ms	19.7 ms
Negate	9.05 ms	133.9 ms	17.5 ms
Median	11.2 ms	2573 ms	477 ms
Binomial	10.6 ms	1084 ms	320 ms